NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

SECURE MOBILE DISTRIBUTED FILE SYSTEM (MDFS)

by

Scott Huchton

March 2011

Thesis Co-Advisors: Geoffrey Xie
Robert Beverly

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From — To)*
24–3–2011	Master's Thesis	2009-03-01—2011-03-31

4. TITLE AND SUBTITLE

Secure Mobile Distributed File System (MDFS)

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Scott Huchton

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Postgraduate School
Monterey, CA 93943

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Department of the Navy

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited

13. SUPPLEMENTARY NOTES

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. **IRB Protocol Number: N/A**

14. ABSTRACT

The goal of this research is to provide a way for frontline troops to securely store and exchange sensitive information on a network of mobile devices with resiliency. The first portion of the thesis is the design of a file system to meet military mission specific security and resiliency requirements. The design integrates advanced concepts including erasure coding, Shamir's threshold based secret sharing algorithm, and symmetric AES cryptography. The resulting system supports two important properties: (1) data can be recovered only if some minimum number of devices are accessible, and (2) sensitive data remains protected even after a small number of devices are compromised. The second part of the thesis is to implement the design on Android mobile devices and demonstrate the system under real world conditions. We implement and demonstrate a functional version of MDFS on Android hardware. Due to the device's limited resources, there are some issues that must be explored before MDFS could be deployed as a viable distributed file system.

15. SUBJECT TERMS

Android, Erasure Code, Shamirs Secret Sharing Algorithm, Java, Mobile Networking

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			19b. TELEPHONE NUMBER *(include area code)*
Unclassified	Unclassified	Unclassified	UU	61	

Standard Form 298 (Rev. 8–98)
Prescribed by ANSI Std. Z39.18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

SECURE MOBILE DISTRIBUTED FILE SYSTEM (MDFS)

Scott Huchton
Lieutenant, United States Navy
B.S., University of Texas at Austin, 1998

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 2011

Author: Scott Huchton

Approved by: Geoffrey Xie
Thesis Co-Advisor

Robert Beverly
Thesis Co-Advisor

Peter J. Denning
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

The goal of this research is to provide a way for frontline troops to securely store and exchange sensitive information on a network of mobile devices with resiliency. The first portion of the thesis is the design of a file system to meet military mission specific security and resiliency requirements. The design integrates advanced concepts including erasure coding, Shamir's threshold based secret sharing algorithm, and symmetric AES cryptography. The resulting system supports two important properties: (1) data can be recovered only if some minimum number of devices are accessible, and (2) sensitive data remains protected even after a small number of devices are compromised. The second part of the thesis is to implement the design on Android mobile devices and demonstrate the system under real world conditions. We implement and demonstrate a functional version of MDFS on Android hardware. Due to the device's limited resources, there are some issues that must be explored before MDFS could be deployed as a viable distributed file system.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

List of Figures

THIS PAGE INTENTIONALLY LEFT BLANK

List of Tables

THIS PAGE INTENTIONALLY LEFT BLANK

List of Acronyms and Abbreviations

AES	Advanced Encryption Standard
API	Application Programming Interface
COTS	Commercial Off the Shelf
DSDB	Directory Service Database
DTN	Delay Tolerant Network
FEC	Forward Error Correction
FIPS	Federal Information Processing Standard
GFS	Google File System
HDFS	Hadoop Distributed File System
JDK	Java Development Kit
JNI	Java Native Interface
JTRS	Joint Tactical Radio System
JVM	Java Virtual Machine
LGPL	Lesser General Public License
MDFS	Mobile Distributed File System
MDS	Maximum Distance Seperable
MLSTP	Multi-Level Security Tunneling Protocol
NDK	Native Development Kit
P2P	Peer to Peer
PGP	Pretty Good Privacy
PKI	Public Key Infrastructure
S3	Simple Storage Solution
SDK	Software Development Kit
VPN	Virtual Private Network
WPA2	Wi-Fi Protected Access 2

THIS PAGE INTENTIONALLY LEFT BLANK

Acknowledgements

First and foremost, I would like to thank my loving and supportive wife, Starla, who's patience with my countless hours on the computer during this thesis process cannot be over appreciated. She endured many evenings of "just a few more minutes" that ended up being many more hours. I would also like to thank my children, Colleen, Maddy, and Kenny, who were every bit as supporting as their mother. Thank you for providing an unending source of entertainment and genuine curiosity during this process. I'm amazed at the people you've become. I couldn't have done this without the support of my family.

I'd like to thank LCDR Chris Rapin, USN for the help with writing advice and listening to my ideas, no matter how dry the content. Our thesis paths were very similar and journeys are always more fun with friends than solo. I'm sure our advisors appreciated that as well.

Finally I'd like to thank my thesis advisors, Geoffrey Xie and Robert Beverly, whose encouragement, guidance, and patience showed me the way and ensured that I completed this thesis. I often felt the journey was similar to a pinball bouncing from one direction to another, but my advisors were always there to mentor, offer encouragement, and keep me moving in the right direction. While not officially an advisor, I would also like to thank John Gibson for the many talks that helped me keep the end goal in focus. Thank you all for the constant support.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1:
Introduction

1.1 Problem

Mobile devices can be harnessed to increase battlefield awareness. Recent research has focused on the ability to provision and field mobile devices to increase communications and situational awareness using next generation tactical radio technologies such as JTRS and mobile networking technologies such as WiFi, 3G, WiMax or LTE. These mobile devices provide a platform for creating mobile networks capable of distributing tactically relevant information to the battlefield [1]. The ability to communicate in almost any environment is ubiquitous. Secure and resilient storage of sensitive data is often vital to effective communication. These mobile devices provide a platform for communication of sensitive information, but at a potential cost to mission and security. This is particularly true in a mobile environment where the underlying architecture is unpredictable. Data can be unreachable due to unreliable communication channels or compromised due to loss or capture of a device.

Suppose a small team of soldiers are on a mission where access to and sharing of data is critical to the success of the mission. The soldiers require a device capable of transmitting and storing sensitive data, but the loss of one or more of those devices could prove devastating if the enemy is able to gain access to the sensitive data. Encryption on the mobile device could solve this problem, but it does not address the issue that the data stored on the lost device is no longer available to the rest of the team. Encryption alone does not solve this problem.

Current approaches to data sharing tend to focus on transmission security. Transmission security addresses one important aspect of the overall security issue, but offers no solution to data resiliency nor data security when a device is captured. This thesis combines several concepts, including Shamir's threshold based secret sharing scheme, erasure codes, and AES encryption, in a novel way to create a functional design for secure and resilient data sharing in a collection of mobile devices. Currently, when a mobile device is lost, either a self-destruct mechanism or a so-called "remote kill" method is required to destroy sensitive data on that device. Both solutions have drawbacks. Self-destruct is a relatively expensive solution. Most COTS devices do not support self-destruct. Remote kill fails if the lost device is no longer connected to the network. This thesis provides an alternative solution. It guarantees that as long as the number

of compromised devices is below a preconfigured threshold, sensitive data is protected. At the same time, the solution supports resiliency by duplicating encrypted fragments of the same data in multiple devices. We propose that data can be stored in fragments across many devices in such a way that only a subset of those devices are required to recover the original file. The key difference between this approach and previous solutions is the use of a one-time session key that requires no pre-sharing and the use of a distributed directory service capable of withstanding the loss of one or more nodes. MDFS requires minimal network infrastructure to function.

In a mobile network, there are two classes of devices: content generators and content consumers. One solution to file distribution is accomplished by complete replication and relies on the content generator to store and distribute files to content consumers. However, the storage of complete files on a device presents security risks if the device is lost or captured. It is either insecure, or requires complex key management schemes.

We propose a Mobile Distributed File System (MDFS) with three components: a distributed directory service, a read function, and a write function. These functions will leverage the storage capacity of the collective devices to securely store store files distributed amongst multiple nodes.

When designing MDFS, we made the following assumptions:

- We assume an external security mechanism such that physical access to the network is considered authorization for access to MDFS. MDFS itself does not provide authentication.
- We assume wireless security, such as WPA2, to prevent an adversary from passively sniffing data transmitted between nodes.
- MDFS is designed to function on a collection of reachable mobile nodes. As a result, we assume the maximum number of nodes to be less than 100.
- Because MDFS will operate on mobile devices with limited resources, we assume file sizes will be less than 5MB in size.

Rather than transmitting complete copies of a file, content generators will encode the file into fragments using MDFS and distribute those fragments in the network. Once the file is stored in the network, meaning all fragments have been successfully stored, the local copy of the file can

be discarded. In order to retrieve the file, a content consumer must gather some subset of the file's fragments to decode the original file.

This approach to network storage offers the following advantages.

- A single device that is no longer connected to the collection of reachable mobile devices does not have the ability to recover any information about the original file as one fragment is less than the minimum threshold required to recover the file. This precludes the need for a remote kill function. If the device is still connected to the network, the remaining nodes can ignore the device. By refusing to respond to the compromised device, the remaining devices prevent the compromised device from obtaining the required threshold number of fragments. This is stronger than simply encrypting data on the device because even if a device operator did somehow know the encryption key, they cannot be compelled to divulge information stored on the phone without the cooperation of the minimum threshold of other devices.

- Even if the device that generated the file is no longer connected to the network, other authorized devices will still be able to recover the file as long as the minimum threshold number of devices are still connected to the network. Authorized content consumers will still have access to the data even if the content creator is no longer attached to the network.

In order to achieve these advantages, MDFS provides two basic primitives: write() and read(). This section provides a brief introduction to the desired functionality. For a more complete description, see Section 3.2.2 and Section 3.2.3.

write

The content generator has some source file, M, to be stored in the distributed storage system. In order to distribute M, the content generator sends M to the `write()` function. The `write()` function generates a session-key, S, and generates ciphertext, C, by encrypting M with S. It then utilizes a Reed-Solomon function, `encode()`, that breaks C into n encoded fragments labeled $c_0..c_{n-1}$. It also uses Shamir's Secret Sharing algorithm to break S into n fragments labeled $s_0..s_{n-1}$. The threshold for recovery of the ciphertext and the key is k. The function returns a set E of encoded fragments where $E = \{(c_0, s_0), (c_1, s_1), ..., (c_{n-1}, s_{n-1})\}$. c_i is a fragment of C and s_i is a fragment of S. Each unique fragment, $(c_i, s_i) \in E$, will be distributed to a device connected to the network.

read

Files can be uniquely identified by a combination of their filename and timestamp. The directory service provides a `list()` function that returns the filename, timestamp, n value, and k value for each file. Section 3.2.1 provides a more details description of `list()`. When a content consumer wants to reconstruct M, it sends the filename and timestamp for M to the `read()` function. The `read()` function can use the filename and timestamp to query the directory service for the addresses of the nodes that hold fragments of M. The `read()` function gathers k', where $k' \geq k$, fragments from other devices to generate the set R where $R \subset E$ and $|R| \geq k$. R contains k' unique Reed-Solomon encoded fragments of C such that $R = \{(c_0, s_0), (c_1, s_1), ..., (c_{k'-1}, s_{k'-1})\}$. The `read()` function utilizes the function `decode(`R`)`. The decode function must first recover C by utilizing a function `decode(`$c_0, c_1, ..., c_{k-1}$`)` to reconstruct C from the FEC encoded fragments. It is also able to generate the decryption key, S, by applying Shamir's secret sharing algorithm to the key fragments distributed with each file fragment. The function decodes C with S to recover M to be returned by the `read()` function. If $|R| < k$, `decode(`R`)` is unable to construct or infer any information about M.

Any fragment, (c_i, s_i), or subset of fragments with size less than k is insufficient to reconstruct M or infer any information about M. Figures 1.1 and 1.2 show an overview of how MDFS functions. In Figure 1.1, Node 1 is the content generator. If $n = 7$ and $k = 4$, Node 1 generate seven fragments and distributes fragments to all nodes, including itself. In Figure 1.2, Node 3 is the content consumer and gathers four fragments from Nodes 2, 3, 4, and 5. Node 1, the content generator, and Node 7 are no longer attached, but Node 3 is still able to gather the minimum threshold number of fragments necessary to reconstruct the file written to MDFS by Node 1. This example omits the directory service details that provide the necessary information to store and retrieve the file fragments.

1.2 Research Questions

How can (k, n) threshold security be used to store files securely in a distributed file system running on a mobile wireless network? The file system must maintain the following properties:

- At least k fragments are needed to recover the original file. Any $k - 1$ fragments yields no information about the original file.
- Once data is written to the network, authorized users will have access to that data even if

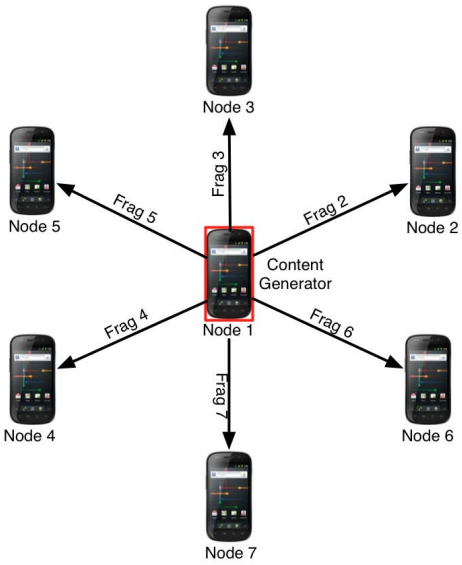

Figure 1.1: MDFS Write ($n = 7$, $k = 4$)

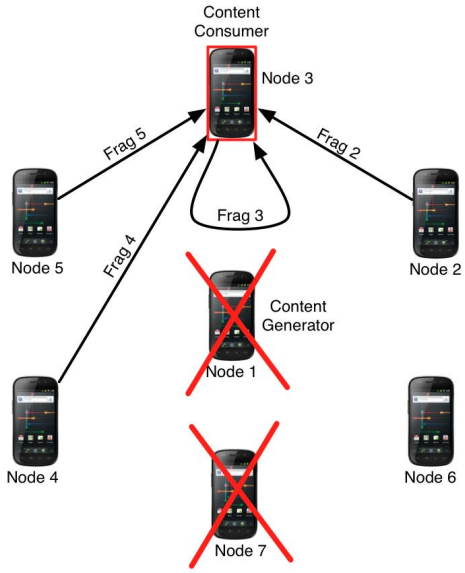

Figure 1.2: MDFS Read ($k = 4$) with loss of two devices

the originator of the file is no longer connected.

- The system is entirely mobile and distributed. It requires no static infrastructure to operate.

MDFS was prototyped in the Java programming language on Android mobile devices running Android 2.2.

1.3 Thesis Organization

Chapter 1: Introduction. This chapter provides a brief introduction to distributed storage along with a very high level overview of required functionality for MDFS. In addition, it outlines the research questions that will be explored in this thesis.

Chapter 2: Background. The background chapter provides a general overview of current technologies that are used for distributed storage. It will compare several similar approaches to distributed storage and explain how they differ from the solution proposed in this thesis.

Chapter 3: Concept and Design. This chapter discusses the design of the thesis topic in detail.

Chapter 4: Implementation and Experimental Results. This chapter discusses the implementation of the design as outlined in Chapter 3 on Android devices. Deviations from concept or design will be discussed.

Chapter 5: Conclusion and Future Work. This thesis topic has many potential uses. This chapter includes possible avenues for future research and improvements to the required functionality.

CHAPTER 2:
Background

2.1 Distributed Storage Basics

The subject of distributed storage has grown in relevance as large data centers emerge to serve storage needs for companies like Google and Amazon. These data centers are typically large buildings with solid infrastructure requirements. Because space and power are readily available, these data centers are able to store data in multiple copies across many servers. For example, Google uses BigTable and MapReduce to store, access, and synchronize multiple copies of data across multiple data centers [2] [3].

There are many approaches to distributed storage. Each approach aims to solve a different problem and each has strengths and weaknesses that make it suitable for the given solution. Table 2.1 provides a quick comparison of different approaches to distributed storage. Four different projects are listed across the top. The vertical axis lists characteristics that can be used to differentiate each implementation. *Encrypted* refers to whether the particular implementation employs encryption before storing fragments of data. *Erasure coded* is a complement to *complete replication*. *Pre-share key* refers to whether the implementation requires a key to be pre-shared before encryption. *Relies on external authentication* refers to whether or not access to fragments or shares is restricted by the implementation or relies on an external authentication mechanism. *Scales to large sizes* refers to the ability of the filesystem to deal with massive storage on the petabyte scale. Bigtable and GFS were designed with type of scaling in mind. Relies on external infrastructure refers to the external components required to implement the distributed storage. Bigtable relies on the most external infrastructure, but both Tahoe-LAFS and Stealth require fixed infrastructure to function.

Google uses a combination of the Google File System (GFS) and Bigtable to achieve their distributed storage needs [2] [4]. GFS is the distributed storage file system and Bigtable provides the means to manage structured data stored on GFS. Google built their infrastructure using commodity components and their primary assumption when designing GFS and Bigtable is that the components will fail regularly. Reliability and availability played a key role when designing and implementing GFS. Google's distributed storage system has many desirable properties, but it relies on many smaller subsystems and is used in large data warehouses storing very large

	MDFS	Tahoe-LAFS	Unisys Stealth	GFS & Bigtable
Encrypted	√	√	√	
Erasure coded	√	√	√	
Complete replication				√
Pre-share key		√	√	
Relies on external authentication	√			√
Scales to large sizes				√
Relies on external infrastructure		√	√	√

Table 2.1: Distributed Storage Comparison

datasets. GFS stores files in fixed-size chunks that are globally identified by a 64-bit chunk handle.

A GFS cluster contains one *master* server and multiple *chunkservers*. The *master* server stores the metadata about the chunks and access control information. The *chunkservers* store the actual chunks which are referenced by their handle. When a client needs access to a chunk, they poll the *master* server for the location of the chunk and then perform their I/O operations directly with the *chunkserver* to prevent the *master* server from becoming a bottleneck. The *master* periodically polls each *chunkserver* with heartbeat messages to give instruction and collect state.

GFS and Bigtable offer many useful insights into how to design a distributed storage file system, but they are clearly designed for a different domain. MDFS must operate with very few resources and does not need to handle very large datasets. Additionally, Google datacenters are tightly controlled and data security relies a great deal on physical security to prevent unauthorized access. MDFS must provide data security at the fragment level since it cannot rely on physical security for access control.

Tahoe-LAFS is an open source project that uses the zfec erasure code library to securely store data on multiple servers [5]. It uses distributed storage to provide "provider independent security" cloud based storage. The goal of Tahoe-LAFS is to provide secure and resilient storage on multiple drives that is resistant to drive failure or malicious attack. A user runs a gateway server on their own network that takes care of encryption and integrity checks before storing erasure coded fragments on external disks. Tahoe-LAFS uses an encryption key stored at the gateway to encrypt the data and then stores erasure encoded fragments of the ciphertext, along

with a hash of the key used to encrypt the fragment on multiple drives [6]. Figure 2.1 provides and overview of Tahoe-LAFS functionality. The Tahoe-LAFS Client sends an unencrypted file via a web API to the HTTPS Server. The HTTPS Server passes the file off to the Tahoe-LAFS Storage client which encrypts the file and then uses erasure coding to store fragments of the file on multiple storage drives.

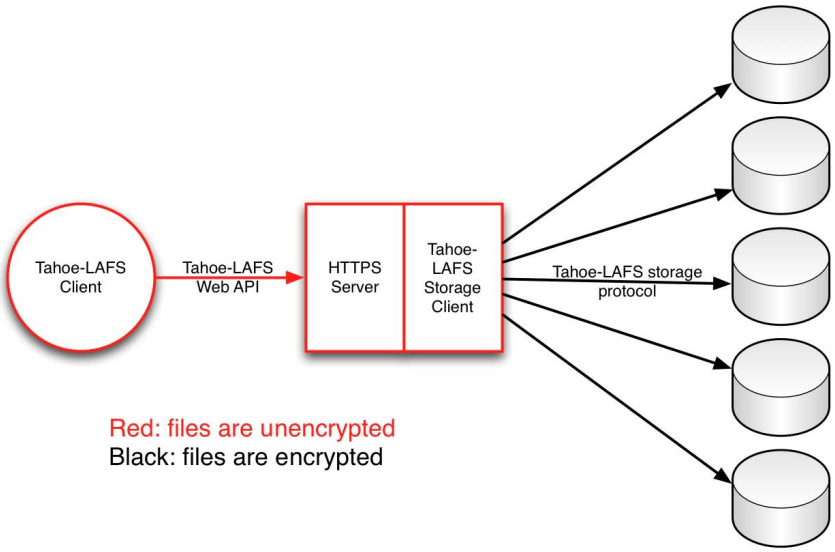

Figure 2.1: Overview of Tahoe-LAFS

Tahoe-LAFS relies on significant infrastructure compared to MDFS, so it is well suited for situations were infrastructure is available. The key difference in approach to distributed storage between Tahoe-LAFS and MDFS is in how the directory service is implemented and how the encryption session key is generated and distributed. MDFS generates a unique session key for each file stored and distributes fragments of the key with fragments of the ciphertext. Tahoe-LAFS relies on a gateway which presents a single point of failure. Tahoe-LAFS mitigates this by making the gateway easy to switch, but MDFS is designed to work without the need for infrastructure. By generating session keys for each encryption and storing the key with the fragments, MDFS eliminates the need to maintain a gateway between the devices accessing the storage medium and the storage medium itself.

While MDFS does not use Pretty Good Privacy (PGP) in its implementation, we explored the combination of symmetric and asymmetric encryption employed by PGP. In order to ensure that individual fragments do not reveal information about M, it is necessary to either encrypt M prior to erasure encoding it, or encrypt the fragments individually after M has been erasure

encoded. Public-key encryption is not a viable option as each fragment will go to different nodes where the destination is not predetermined. Key management would be complex in this scenario. Additionally, public-key encryption has several orders of magnitude higher costs than symmetric encryption making it a less desirable solution on resource constrained mobile devices. Symmetric encryption has significant performance advantages over asymmetric encryption, but each node would have to maintain a copy of the encryption key, which would be insecure if one of the nodes were compromised.

Pretty Good Privacy (PGP) uses a combination of public-key and conventional encryption to provide security services for electronic mail messages and data files [7]. It works by generating a session key and symmetrically encrypting mail or data. Then it uses the public key of the recipient to encrypt the much smaller session key and sends the symmetrically encrypted message along with the privately encrypted session key. PGP takes advantage of the relative efficiency of symmetric encryption while providing the benefits of public-key encryption. MDFS adapted adapted PGP's use of a session key to symmetrically encrypt data, but uses a different method to distribute the session key that doesn't employ PKI.

Towards the end of our implementation phase, we became aware of another very similar distributed storage solution system called Stealth Technology. Stealth was introduced by Unisys in 2009. Unisys has taken an approach similar to MDFS for distributed storage, but their solution aims to solve a completely different problem. Stealth uses distributed storage and encryption to allow an organization to simulate multi-level security access based on workgroup affiliations. Stealth uses the Multi-Level Security Tunneling Protocol (MLSTP) at a gateway to break data into fragments and then encrypts those fragments using a session key before storing the fragments in cloud storage. Encryption is accomplished using a technology called SecureParser which distributes the key as fragments with each share of data similar to the way MDFS stores the key with its fragments. Stealth is similar to MDFS in that access to the shares acts as the authority to view the data. However, MDFS is implemented on relatively constrained resources and does not rely on gateway servers or tunneling protocols to achieve security. MDFS's ability to operate without infrastructure in a wireless environment differentiates it from Stealth.

MDFS's primary differentiation from other implementations is its distributed architecture that relies on very little static infrastructure. MDFS is designed to meet the challenges of both security and resiliency in a mobile environment by implementing a robust file and fragment management system capable of storing and retrieving fragments on mobile devices as needed.

2.2 Advanced Encryption Standard (AES)

AES is a Federal Information Processing Standard (FIPS) approved cryptographic block cipher standard used to encrypt and decrypt electronic data. It processes 128 bit data blocks using cipher keys with lengths of 128, 192, or 256 bits. AES encryption takes a plaintext message and a key to encrypt the plaintext message into an unintelligible stream called ciphertext. AES decryption take the ciphertext and the same key to decrypt the ciphertext into the original plaintext message [8]. A Java implementation of AES is included with the Java SDK as part of the Java Cryptography Extension.

2.3 Erasure Codes

Erasure codes are forward error correction (FEC) codes that translate some message M of length $|M|$ into a coded message with a length greater than $|M|$ such that M can be recovered from some subset of the coded message. In 1960, Reed and Solomon introduced a Maximum Distance Separable (MDS) algorithm [9]. An MDS erasure codes stores a message M in n fragments of size $|M|/k$ such that any k fragments is sufficient to reconstruct M [10].

Figure 2.2 shows an example of an erasure coding where $n = 5$ and $k = 3$. The original file is encoded into 5 fragments. Fragments 3 and 4 are lost, but fragments 1, 2, and 5 remain. Since the threshold for recover is 3, there are sufficient fragments remaining to recover the original file.

Erasure codes have the desired threshold recovery property, meaning that k or more fragments out of n enable the reconstruction of M, but they do not offer security of the individual fragments. One of the requirements of MDFS is that any fragment m_i or subset of fragments less than k is insufficient to reconstruct M or infer any information about M. Unfortunately, some of the fragments generated by Reed-Solomon encoding are just blocks of the original data. Therefore, if an adversary were to capture some of the fragments, they would be able to infer partial information about M, which does not meet the requirements of the system.

The use of erasure codes for storage resiliency is not new. The Apache Hadoop Disk File System (HDFS) project has implemented a plugin to allow the use of erasure codes to reduce disk storage requirements. The HDFS specification requires the filesystem to keep three copies all data. However, a patch was introduced to HDFS to implement erasure codes that effectively maintain a replication factor of three while reducing disk storage requirements to 2 [11]. Erasure codes support the resiliency requirement for MDFS, but do nothing for the security requirement.

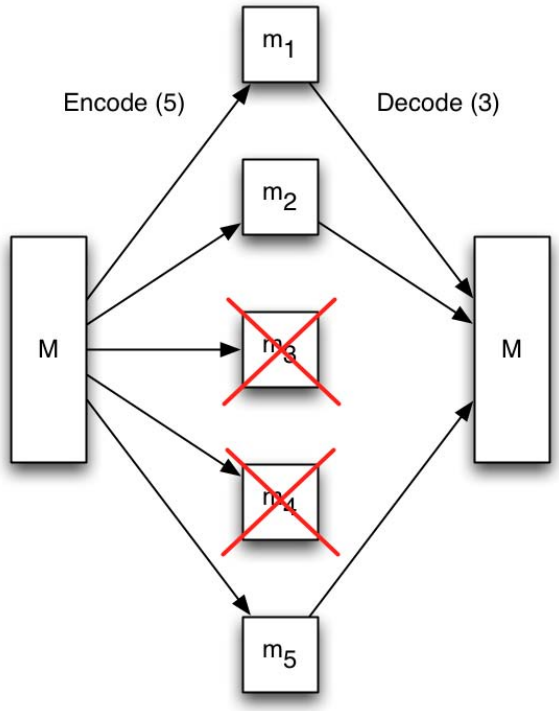

Figure 2.2: (3, 5) Erasure Code

2.4 Shamir's Secret Sharing Algorithm

Fragment destinations are not predetermined, so public key management would be incredibly difficult. Public key dissemination and management between arbitrary nodes is both a complex process and a likely point of failure. Furthermore, it would be an undesirable side effect to require the receiving node to decrypt the fragments and store them unencrypted on their physical memory. We would like to take advantage of the efficiency of symmetric encryption, without the complication added by public-key distribution.

Shamir's Secret Sharing Algorithm is closely related to Reed-Solomon and other MDS erasure codes [12]. First discussed by Adi Shamir in 1979, he explains how to distribute information using threshold security. In his example, he describes a situation where a company, with multiple authorized executives, would require at least three signatures to sign a corporate check. This system is an example of $(3, n)$ threshold security.

Shamir's algorithm has the properties of security and resiliency required of MDFS, but is not

efficient. It suffers from the fact that each fragment is larger than the original message. This would increase network activity to unacceptable levels for relatively small files.

2.5 Putting it Together

By replacing the public key step of PGP with Shamir's Secret Sharing Algorithm, we can generate a session key and encode it using the same n and k values as used for the Reed Solomon encoding. We can generate a session key, encrypt M to generate the encrypted file C, encode n Reed Solomon encoded fragments of C, generate n key fragments using Shamir's algorithm, and distribute n fragments containers with a file fragment and a key fragment each.

When k fragment containers are assembled, the node has k file fragments and k key fragments which is sufficient for recovery. The node can decode the key, S, using the key fragments. It can erasure decode the file fragments to recover C. and then use S to decode C and recover M where $M = AES_S(C)$.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 3:
Concept and Design

3.1 Concept

MDFS targets both the secure storage and the secure dissemination of data. In a mobile ad-hoc network, there are two classes of devices: content generators and content consumers. One implementation of distributed storage is accomplished by storing multiple complete replications and relies on significant infrastructure to organize and retrieve data. Google's Bigtable is an example of this implementation. One common distribution implementation relies on the source to store and distribute files to receivers [2]. Twiddlenet is a mobile content sharing system developed at Naval Postgraduate School to facilitate rapid dissemination of information between first responders in a crisis [13]. It accomplishes this task by allowing content consumers to subscribe to content creators via atom feeds. When a content consumer sees a file in an atom feed, it sends a request to the content creator and a traditional file transfer is initiated. Twiddlenet provides a convenient means of sharing data in a mobile environment, but it lacks security features and redundancy necessary to secure the data.

Storing multiple copies of data is not practical with the limited storage available to mobile devices. Relying on the content generator to provide copies of data to each requestor does not meet the resiliency requirements of MDFS. In order to address these issues, we propose a Mobile Distributed File System (MDFS) that leverages a robust directory service, AES symmetric encryption, erasure coding, and Shamir's secret sharing algorithm to achieve security of data at rest while providing resiliency, maintaining usability, and reducing end-user complexity.

3.1.1 Requirements

Security

One of the primary requirements of MDFS is that data at rest is secured against unauthorized disclosure. Storage of complete sensitive files on the device constitutes a security risk if the device is in the physical possession of an adversary.

In order to address this requirement, MDFS uses 256 bit AES encryption and employs a (k, n) threshold scheme [12]. MDFS takes some file M and divides it into n pieces $m_0, m_1, ..., m_{n-1}$, where $m_i = (c_i, s_i)$, such that:

1. M can be easily computed with knowledge of any k (where $1 < k \leq n$) or more m_i fragments

2. M is completely undeterminable with knowledge of any $k-1$ or fewer m_i fragments

In order to recover M, a node requires the cooperation of at least k nodes. Since any $k-1$ nodes leaves M completely undetermined, the requirement that no information about M can be determined by analyzing any one fragment is also satisfied.

Resiliency and Efficiency

Resiliency addresses the requirement that a lost node does not prevent other authorized nodes from accessing data. The storage of complete files on any one mobile device means that all content created by that mobile device is gone if it is lost or destroyed. By adopting Shamir's convention of setting $n = 2k-1$ to satisfy the security requirement as discussed in Section 3.1.1, we are able to lose up to $k-1$ nodes while still maintaining the ability to recover M.

Usability

The filesystem is designed as a low level library that exports a simple API to store and retrieve files. The complex task of fragment generation and tracking is taken care of by MDFS's directory service so the application user is not exposed to fragment management. The API is simplified such that the `write()` method takes a filename, a n value, and a k value as arguments. The `list()` function requires no arguments. The `read()` function takes a filename and timestamp as arguments. The filename and timestamp arguments can be retrieved from the directory service's `list()` method.

3.2 Design

Each fragment is packaged in a fragment container (Figure 3.1) that contains the fragment bits along with metadata required by the erasure coding library and the Shamir coding library. The fragment container contains a method to generate a unique MD5 fragment hash based on the filename, creation timestamp, and the fragment number to identify individual fragments within a file. It also provides a method to generate a file hash based on just the filename and creation timestamp. This allows MDFS to ensure that fragments with different fragment hash values are from the same file. In other words, fragments with the same file hash value, but a different fragment hash value are unique fragments of the same file.

We chose to use the file's timestamp in the hash rather than some other metadata for a number of reasons. The original motivation was that it was readily available for all files and the likelihood of two different files having the same filename and the exact same timestamp is remote. It provides an easy way to differentiate between two files with the same filename. Furthermore, it allows for future development where files could be updated and versioned based on their timestamp. In other words, if a file is updated, it would have a more recent timestamp than the original. If the updated file's metadata contains a reference to the original file hash, then file versioning over time could be maintained. Figure 3.2 shows how one file could be updated with references to the original for versioning. As our initial design of MDFS does not include versioning control, this implementation has not been designed into MDFS, but is intended to explain our motivation for using the timestamp in the hash function over other metadata.

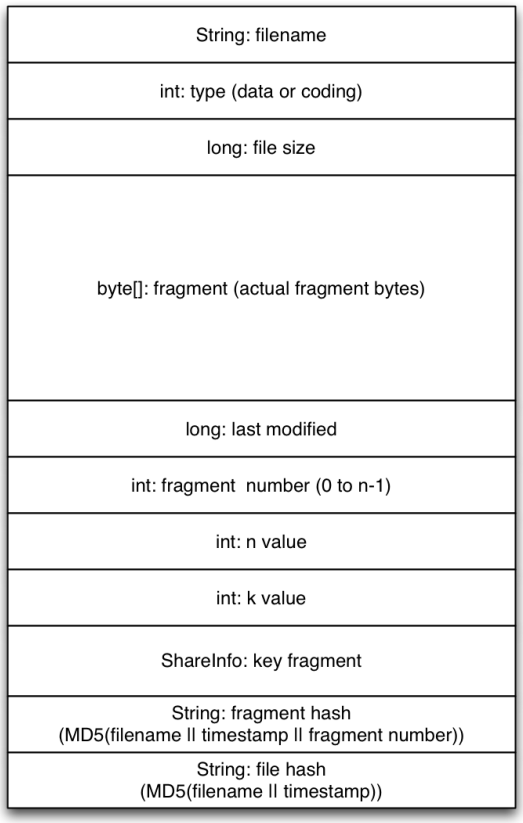

Figure 3.1: Fragment Container

The fragment container class provides methods, `getFragmentHash()` and `getFileHash()`, to obtain the hash values of file container objects. Additionally, the fragment container class

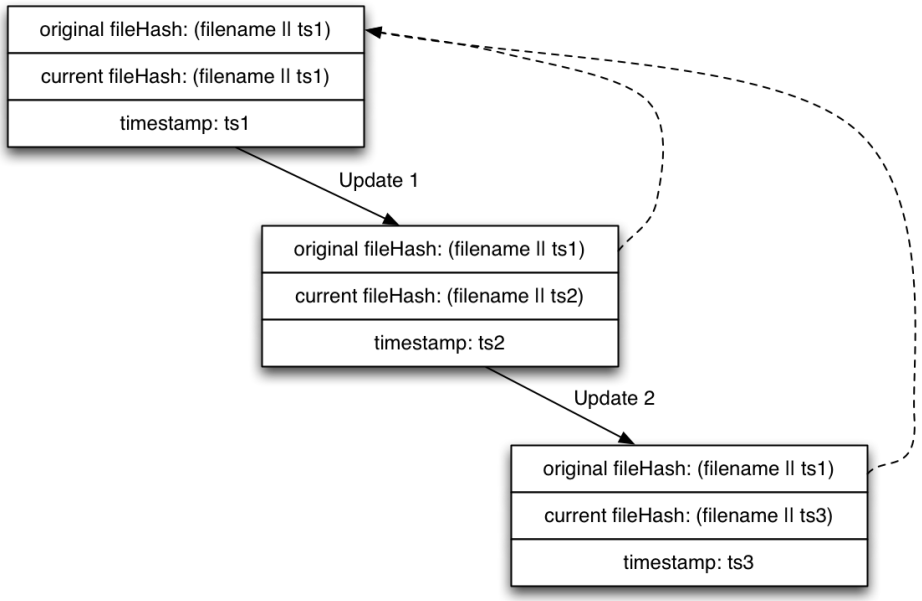

Figure 3.2

provides static methods, `generateFragmentHash(filename, timestamp, fragNum)` and `generateFileHash(filename, timestamp)` to generate the hash values given the filename, timestamp, and fragment number for use in lookup operations. This hash scheme forms the basis for how the directory service manages files and how the `read()` and `write()` functions use the directory service to access fragments.

The proposed file system contains three major design components for mobile file distribution: read, write, and a directory service. During a `write()`, for each fragment successfully stored on a remote node, the content generator registers the fragment hash value and its destination address with the directory service. When all fragments have been successfully written, the content generator registers the filename, timestamp, source, n, and k values as metadata with the directory service.

A content consumer can poll the directory service for a current list of files and then request one of those files. When the content consumer requests a list of files, the directory service will return a list containing filenames, timestamps, source description, n, and k values. Given the filename, and timestamp, the `read()` function can use the static methods from the fragment container class to generate fragment hash values from $0 \rightarrow n - 1$ and request the location of

those fragments from the directory service. The content consumer can then retrieve k fragments from other nodes and recover M.

3.2.1 Directory Service

The directory service is the foundation for the other pieces of MDFS to function properly. In order to securely encode and decode distributed file fragments that meet the security requirements outlined in Section 3.1.1, it is necessary to create a robust directory service capable of managing the fragments.

The directory service is a distributed information store that is referenced by the `write()` function to register files and fragments. It keeps track of which files are written to MDFS and where each fragment associated with each file is located. The `read()` function uses the directory service to request a list of files stored in MDFS and to find the location of fragments associated with a particular file. It is important that the directory service be distributed and sufficiently redundant to avoid becoming a single point of failure for the entire system.

Each node will maintain its own directory service database (DSDB). This database consists of three main tables. The **files** table holds the filename, timestamp, source description, n, and k values for each file present in the system. The **fragments** table holds the fragment hash value and address of the node that holds that fragment for each fragment. Finally, as each node connects to the network, it will register as an active node in the directory service. The directory service will maintain a list of neighbors and will respond with a list of n addresses when queried for storage locations by the `write()` function. Identical copies of the database are maintained at each node. Updates to the database are accomplished using multicast flooding to all nodes listening on a multicast channel.

Directory Service Methods to Support the Write Function

As the `write()` function executes, the content generator is responsible for updating the directory service with information about the fragments and the file. Once the `write()` function has successfully completed, the directory service will have an updated file list containing an entry for the new file and the address where each fragment associated with that file is stored. The following functions provide this functionality.

`void registerFragment(String fragHash, String location)`

After each successful transfer of a fragment, the content generator will register the fragment hash value and storage location with the directory service. The directory service will store this

information in a database so that it can provide the location of a fragment given its fragment hash value during the `read()` method.

`void registerFile(String filename, long timestamp, int n, int k, String source, String[] filterKeywords)`

Once all n fragments have been written to remote nodes and registered with the directory service, the content generator will finish the `write()` function by registering the filename, timestamp, n value, k value, source, and keywords with the directory service for inclusion in its list of files.

The `filterKeywords` parameter is optional array of strings and allows the user to define keywords that can be used to search the directory service. If no keywords are provided, it defaults to the source.

`String[] getActiveNodes(int n)`

The *write()* function requires a list of available active nodes. As each node joins the network, they register with the directory service as an available active node. This function returns a list of node addresses corresponding to n active node entries. The nodes returned by this function are selected such that fragment storage is balance across all active nodes.

Directory Service Methods to Support the Read Function

`DataRecord[] list(String[] filterKeywords)`

This method returns an array of DataRecords containing the filename, creation timestamp, filter keywords as a String array, n, and k values for each file that has been registered with the directory service. The source description is for information purposes only and is not required to retrieve the fragment.

The `filterKeywords` parameter is an optional list of strings and allows the user to pass a parameter to the `list()` function to filter the results. The strings in `filterKeywords` will be compared to the strings in `filterKeywords` in the DataRecord object for each file and only return matching results. If the user provides no keywords, the `list()` function returns all file entries.

`String whoHasFragment(String fragmentHash)`

Once the content consumer has selected a file from the list of files provided by the directory service, it can provide the filename, timestamp, and fragment number to the static methods of the fragment container in order to generate fragment hash values that can be used to request the

location of file fragments. The directory service provides the address of the node that has the fragment so the content consumer can request the fragment directly from the active node.

3.2.2 Write Function

1. The sender has some file M to distribute
2. The sender picks the number of fragments to generate, n, and the threshold for recovery, k
3. The sender generates a random session key, S, for symmetric AES encryption of this file only
4. The sender symmetrically encrypts file M with S to generate an encrypted file, C where $C = AES_S(M)$
5. The sender encodes the encrypted file C into (k, n) fragments using Reed-Solomon erasure encoding resulting in $\{c_0, c_1, ..., c_{n-1}\}$
6. The sender encodes S into (k, n) fragments using Shamir's secret sharing algorithm resulting in $\{s_0, s_1, ..., s_{n-1}\}$
7. The sender packages file fragments and key fragments to create n fragment containers such that $\{(c_0, s_0), (c_1, s_1), ..., (c_{n-1}, s_{n-1})\}$
8. The sender requests the address of n available active nodes from the directory service
9. The sender distributes fragment containers to n neighbors and deletes the local complete copy of M

In order for the write to be successful, n fragments must be written to the system. The content generator will request n addresses from the directory service. The content generator creates an integer array `success[n]` and initialized to all 1s to keep track of which fragments have been successfully written to the system. An integer variable `fragmentRemaining` is initialized to n to keep track of how many fragment still need to be transmitted. The content generator writes the fragments, contained in a fragment vector from $0 \rightarrow (n-1)$, to other nodes. If it is successful, `success[current_index]` is set to 0 to indicate the fragment has been successfully transmitted, the fragment hash value and destination address are registered with the directory service, and `fragmentsRemaining` is decremented by 1. The write loop continues

until `fragmentsRemaining` $= 0$, indicating that all fragments were written successfully. This process is shown as pseudocode in Figure 3.3.

```
WRITE(byte[] file, DataRecord record, String[] neighbors, String[] filterKeywords)
 1  int index = 0
 2  int fragmentsRemaining = n
 3  DirectoryService ds = DirectoryService.getInstance()
 4  fragments[n] = MDFS.getFragments(File, n, k)
    // returns a Vector<FragmentContainer> of size n
 5  let success[0..n − 1] be an array of 1s
 6  while fragmentsRemaining > 0
 7      if success[index] == 1 // only send if haven't sent before
 8          send(fragment[i], neighbors[i])
 9          if send == success
10              success[index] = 0
11              ds.registerFragment(fragments[i].getFragHash(), neighbors[i])
                // registers the fragment hash value and the destination
                // address with the directory service
12              fragmentsRemaining = fragmentsRemaining −1
13      index = (index + 1) MOD n // loops from 0 to n − 1
14  ds.registerFile(record.filename(), record.timestamp(), record.n(), record.k(), filterKeywords)
```

Figure 3.3: `write()` Pseudocode

This code assumes that the neighbors will be accessible. If not, then it is possible for this loop to continue indefinitely. In order to mitigate this issue, the content generator could request `fragmentsRemaining` new addresses when `index` $= n$. It is also important to note that the write could be considered successful when any number of fragments greater than k have been successfully sent, but any number of fragments less than n reduces reduces resiliency.

The interesting part of this pseudocode is the `getFragments` function call. This is the function that actually performs the encryption, erasure coding of the file, and Shamir encoding of the AES key. Details of how this function works are addressed in the pseudocode in Figure 3.4.

To maintain security, MDFS should ensure that one host does not contain more than one fragment from each file, or if that is not possible, no host has k or more fragments.

The final step is for the content generator to update the directory service with the filename, timestamp, source, n value, and k value of the file so that the directory service is able to provide

```
Vector<FragmentContainer> GETFRAGMENTS(byte[] file, int n, int k)
 1  FragmentContainer tempContainer = null // temporary holder
 2  Vector<FragmentContainer> fragmentVector = new Vector<FragmentContainer>()
 3  AESKey = generateAESKey()
 4  byte[] encryptedFile = AESEncrypt(file, AESKey)
 5  byte[][] fileFragments = ReedSolomon.encode(encryptedFile, n, k)
 6  byte[][] keyFragments = ShamirSecretShare.encode(AESKey, n, k)
 7  for i = 0 to n − 1
 8      tempContainer = new FragmentContainer(fileFragments[i], keyFragments[i])
 9      fragmentVector.add(tempContainer)
10  return fragmentVector
```

Figure 3.4: `getFragments()` Pseudocode

an updated list of available files to any requesting node.

3.2.3 Read Function

1. The receiver wants to recover some file, M, obtained from the directory service `list()`.
2. The receiver requests at least k unique fragments of M from neighbors until it has $\{(c_0, s_0), (c_1, s_1), ..., (c_{k-1}, s_{k-1})\}$
3. The receiver reconstructs S from $\{s_0, s_1, ..., s_{k-1}\}$ using Shamir's secret sharing algorithm
4. The receiver recovers the encrypted file, C, by applying the Reed-Solomon erasure decoder on $\{c_1, c_1, ..., c_{k-1}\}$
5. The receiver recovers M by applying S to decrypt C such that $M = AES_S(C)$

In order for the read to be successful, at least k fragments must be gathered from the system. The content consumer can query the directory service for a list of available filenames and the corresponding timestamp, n value, and k value for each. Using the `generateFragmentHash()` method of the fragment container class, the content consumer can generate the hash for each fragment of the desired file. It then uses that hash as the argument to the `whoHasFragment()` method of the directory service to find the location of each fragment. This process is shown as pseudocode in Figure 3.5.

```
byte[] READ(String filename, long timestamp, int n, int k)
 1  byte[] plainByteFile[] = null // holds the decoded file in bytes
 2  index = 0
 3  totalFrags = 0
 4  FragmentContainer fragment = null // temp holding value
 5  DirectoryService ds = DirectoryService.getInstance()
 6  Vector<FragmentContainer> fragments = new Vector<FragmentContainer>()
 7  String fragHash = null // temp holder to hold the calculated fragment hash
 8  while totalFrags < k
 9      fragHash = FragmentsContainer.generateFragmentHash(
            filename, timestamp, index))
10      address = ds.whoHasFragment(fragHash)
11      fragment = getFragment(fragHash, address) // returns the FragmentContainer
12      if getFragment == success
13          fragments.add(fragment)
14          totalFrags = totalFrags +1
15      index = index +1
16      if index == n throw InsufficientFragmentException
    // fragments Vector now contains k FragmentContainers
17  plainByteFile = MDFS.getFile(fragments, n, k)
    // returns the File as a byte array
18  return plainByteFile
```

Figure 3.5: `read()` Pseudocode

The content consumer will start at fragment 0 and will request fragments until it has obtained k fragments, which is sufficient for the recovery of M. If the content consumer reaches n requests before obtaining k fragments, there are insufficient fragments available to recover M. This error condition occurs if $(n - k) - 1$ nodes are unreachable.

As with the `write()` function, the interesting part of this pseudocode is the `getFile()` function call. Once k fragments have been obtained, MDFS is able to reconstruct the encryption key, reconstruct the encrypted file, and decrypt the file using the encryption key to recover M. This is the function that actually performs the combining of fragments, the combining of the AES key fragments, and the decryption of the file. Details of how this function works are addressed in the pseudocode in Figure 3.6 on page 25.

```
byte[] GETFILE(Vector<FileContainer> fragments, int n, int k)
1  byte[] encryptedByteFile = null
2  byte[] plainByteFile = null
3  for int i < fragments.size()
4       process fragments
5       process keyFragments
6  encryptedByteFile = ReedSolomon.decode(fragments)
7  AESkey = ShamirSecretShare.decode(keyFragments)
8  plainByteFile = AESDecrypt(encryptedByteFile, AESKey)
9  return plainByteFile
```

Figure 3.6: `getFile()` Pseudocode

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 4:
Experimental Results

4.1 Equipment Used

In just the past two years, there has been a marked increase in availability of smart phones. Apple's iPhone, running iOS, greatly increased the demand for smart phones with a corresponding App Store that allows consumers to easily install third party applications. The iPhone was quickly joined by several models of phones running Google's Android OS capable of installing third party applications via the Android Market. Both iOS and Android devices provide more than enough capability to implement network applications. Given the wide variety of phones available for implementation, it was necessary to choose one technology for implementation purposes.

Android devices provide an easy path for "on device" development without the need to join a developer program. Another differentiating factor between Android devices and iOS devices is that Android devices are programmed using the Java programming language while iOS devices use Objective-C. Given the similarity in device capability, we chose Android due to its availability, capability, and our familiarity with the programming language. As of March, 2011, Android holds the largest market share of smartphones in the United States [14]. We implemented MDFS on HTC Evo 4G phones running Android 2.2.

4.2 Device Limitations

Mobile application development presents some restrictions that are not encountered in typical development scenarios. As mentioned before, Android development is done in Java. However, not all libraries and features of the standard Java Development Kit (JDK) are available in Android. Specifically, some libraries are modified to provide efficiency on devices with limited memory and power, or modified to accommodate the additional security requirements of a cell phone. For example, available heap memory and access to disk storage is severely limited in Android when compared to programming using the standard JDK.

One of the most challenging aspects of developing MDFS was porting the JErasure 1.2 library, written in C, to run on Android. The standard JDK provides the Java Native Interface (JNI) to allow native code libraries to run in the JVM. Android provides an analogous development

environment called the Native Development Kit (NDK). The NDK allows an application developer to compile code written in C into native libraries that can be called by the Java program. The NDK was on revision 3 when we began to develop MDFS and the documentation for implementing native libraries on Android was very limited. Each revision of the NDK adds significant functionality, documentation, and support for additional features. NDK revision 3 did not include build scripts, so we spent a significant amount of time learning how to build native libraries that would run on Android. Google released NDK revision 4 in June 2010 which introduced a build script and significantly simplified the build process for native libraries. In December 2010, Google released NDK revision 5 which added significant features that will be useful for follow on work that requires native library development [15].

MDFS relies on the availability of other devices to function correctly. Each device acts as both a server and a client. In traditional server/client applications, the server is always listening to the server socket and available for connections. Mobile devices do not listen to the server socket when they are sleeping. Furthermore, each application runs as a separate process in Android, so unless the application is actually running in the foreground, the phone will not respond to server requests from other devices.

Android provides a mechanism called a Broadcast Receiver, which allows a program to register to receive notifications when certain events, called Broadcasts, occur on the phone. We were able to mitigate the problem of requiring the application to be running in the foreground by writing our own service to act as a daemon. This service registers to receive a Broadcast when the phone boots and responds by listening on the server socket while the phone is on. When a connection is made from a client, the service processes the message in the background. This approach solves the problem of requiring the application to run in the foreground to function correctly, but the service will not respond while the phone is in sleep mode. While it is not required that the MDFS application be running, it is required that the phone be powered on. It is possible to set the phone to prevent it from going to sleep, but that has significant impact on battery performance.

4.3 Application Details

4.3.1 Jerasure 1.2

A number of excellent erasure code software libraries are available as open source projects. Two of the most common are zfec and JErasure [16]. Erasure code libraries are written in C

for efficiency reasons. We considered porting one of the implementations to Java to examine the feasibility of performing erasure coding using Java on Android hardware. At the time, documentation of the NDK was minimal and it wasn't clear if a C implementation of an erasure code library would compile under the NDK. Both zfec and JErasure use pointers extensively in their implementation. Porting C code that relies on pointer math to Java is not trivial, so we chose to experiment with the NDK.

JErasure is written in C and zfec is written in a combination of C and Python. Plank conducted experiments and concluded that zfec had the fastest Reed-Solomon implementation and JErasure was fastest in all other erasure code implementations [16]. After examination of functionality and source code, we chose JErasure 1.2 for our Reed-Solomon erasure coding library due to its flexibility and it's pure C implementation. MDFS implements classic Reed-Solomon coding, but JErasure provides many more erasure coding algorithms, which could allow for more options and flexibility in future work.

4.3.2 secretsharejava

MDFS utilizes a library called secretsharejava for its Shamir Secret Sharing algorithm [17]. This library is a Java implementation of the LaGrange Interpolating Polynomial Scheme as described in Applied Cryptography [18, 528].

Both the secretsharejava and the JErasure 1.2 libraries are released under the GNU Lesser General Public License 2.1 (LGPL) which allows for them to be used in MDFS in accordance with the LGPL.

4.4 Implementation Details

4.4.1 Source Code

Figure 4.1 on page 30 shows the directory structure of the SecureShare application. The `jni` directory contains the source code for the native JErasure library. The `libs` directory contains the dynamically linked JErasure library compiled for Android devices. The `src` directory holds the Java source code with a base package of `edu.nps.secureshare`. All Android specific code resides in the `android` subdirectory. The `crypto`, `directoryservice`, `network`, `mdfs`, `reedsolomon`, `shamir`, and `util` directories contain code corresponding to their respective function capable of running under a standard Java Virtual Machine (JVM) and is not specific to the Android platform. All code under the `shamir` subdirectory is part of the secretsharejava library.

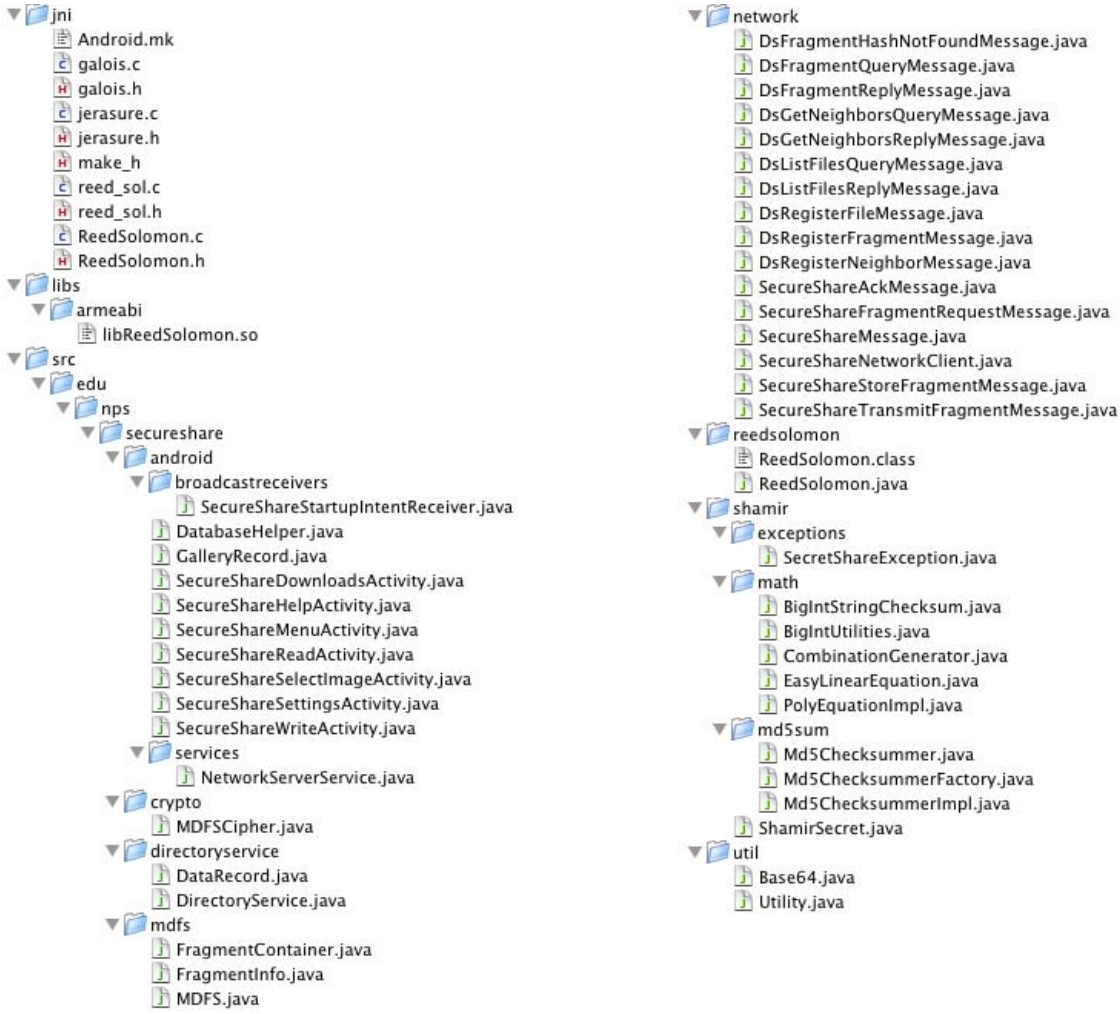

Figure 4.1: Tree View of SecureShare Source Code

4.4.2 Directory Service

The largest deviation between design and implementation occurs in the directory service. The design calls for a completely distributed directory service to avoid a single point of failure. Due to time constraints, the actual implementation utilizes a single node as the directory service. All other nodes communicate with the node acting as the directory service via unicast messages. The node acting as the directory service maintains a database to fulfill directory service queries.

This implementation does introduce a single point of failure for the directory service, but allowed the experiments to be conducted without implementing an additional multicast networking service. The messages used to communicate with the designated directory service node are

the same messages that would be used on a multicast channel to update the directory service. Each installation of the SecureShare Android app is exactly the same on all nodes, so there is no modification to the underlying architecture of the software.

4.5 Experimental Setup

The test bed consisted of five Android devices networked in infrastructure mode using a wireless access point. The use of infrastructure mode is necessary because current Android implementations do not support Mobile Ad-Hoc network without modification of the platform itself. One phone was chosen arbitrarily to act as the directory service.

Description	IP Address
Node 1	192.168.1.100
Node 2	192.168.1.101
Node 3	192.168.1.102
Node 4	192.168.1.103
Node 5	192.168.1.104

Table 4.1: Experimental Setup: Notional IP Address Assignment

For each case, one node acts as the content generator. An image file will be encoded with MDFS using $n = 5$ and $k = 3$ in accordance with the convention that $n = 2k-1$. One fragment will be distributed to each of the 5 nodes. Each fragment may be inspected to ensure that no information about M, other than the public information such as filename, n, and k, is determinable given just one fragment. Table 4.1 will be used as a notional IP address assignment to describe the experiments and their results.

Each experiment was run from a completely fresh installation of the SecureShare application to ensure that there was no artifact data remaining from previous runs. One node is always available to act as the directory service. When SecureShare is launched, the user is prompted to enter the IP address of the node to act as the directory service. All five nodes use the same IP address. Once the IP address is entered, the node registers with the directory service as an active node and is now available to the `getActiveNodes()` function provided by the directory service. Because all messages must pass through the directory service node, we use this node to monitor network activity and application logging.

For each experiment, Node 1 is designated the directory service. Node 2 is designated the content generator. Using the camera on Node 2, we took a picture and selected it to act as

the file, M. Node 2 requested five active node addresses from the directory service. Only one fragment store message was observed while monitoring the directory service at Node 1 as the other fragments were sent directly to the remaining four nodes. After each fragment was transferred to a neighbor, Node 2 registered the fragment hash value and the destination address with the directory service. After all fragments were registered, Node 2 registered the filename, n value, k value, and source address with the directory service. Once the file was registered with the directory service, we verified that each node could see the file available for download by calling the `list()` function of the directory service from each node (Figure 4.2).

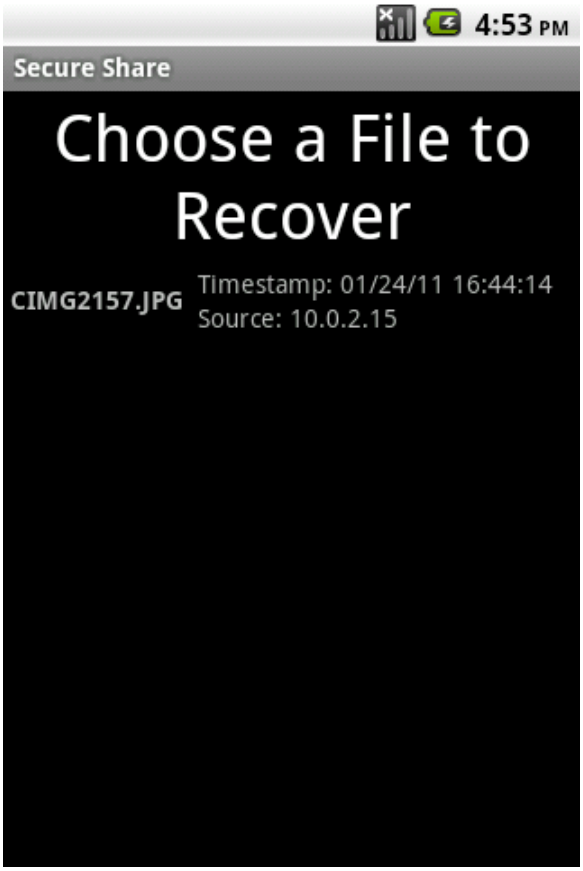

Figure 4.2: View of `list()` output with one file

4.6 Experimental Results

4.6.1 Show that M is recoverable given the loss of any 2 nodes

Nodes 3 and 4 were switched off so that they were no longer listening for fragment request queries. Node 2 selected the image from the available downloads listing. The `read()` function

requested fragment locations for fragments in order from $0 \rightarrow n-1$. Five `whoHasFragment (fragHash)` queries were observed at the directory service as expected. Node 2 needed three fragments to recover M, so it queried and received fragment locations for nodes 1 and 2. The directory service returned the address for nodes 3 and 4, but those nodes were not available, so the third fragment was retrieved from node 5. With three fragments, Node 2 was able to recover and display M.

4.6.2 Show that M is recoverable given the loss of the content generator

Node 2 was switched off so that it was no longer listening for fragment request queries. Node 4 selected the image from the available downloads listing. Four `whoHasFragment(fragHash)` queries were observed at the directory service as expected. Node 4 needed three fragments to recover M, so it queried and received a fragment location from node 1. Node 2 did not respond to the fragment query and nodes 3 and 4 returned fragments. With three fragments, node 4 was able to recover and display the image that was uploaded by node 2 despite the fact that node 2 is no longer in the network.

4.6.3 Show that M is not recoverable if less than k phones are available

Nodes 3, 4, and 5 were switched off. Since only two nodes remain, which is less than k, we expected that M would not be recoverable. We observed five `whoHasFragment(fragHash)` messages at the directory service as expected. Passing less than k fragments to MDFS for recovery of M causes SecureShare to crash. While it would be desirable for SecureShare to handle the exception more elegantly, node 2 was unable to recover M when less than k devices were available.

These experimental results demonstrate that MDFS is capable of meeting the requirements as discussed in Section 3.1.1. However, this implementation suffers stability problems related to heap size and allocation. If the heap grows larger than what is allowed by Android, the application crashes. Further refinement of the implementation may mitigate this problem, but resource constraint will remain a limiting factor. This issue is discussed further in Section 5.3.

4.6.4 Security Analysis

Erasure coding, without encryption, does not provide security for each fragment. In fact, the first k fragments are unencoded fragments of M with size $|M|/k$. If a human readable 900 byte text file is erasure encoded with $n = 5$ and $k = 3$, the first 3 fragments would be 300 byte readable text files. By using AES encryption before erasure coding, the files are unreadable.

The MDFS fragments are stored on the Android device in external storage (Figure 4.3). Some public information required for the erasure code and Shamir's algorithm to function, such as filename, n value, and k value, are prepended to the encrypted data and is viewable by inspecting the fragment files with a HEX Editor. No information, beyond the public information, is discoverable by inspecting the fragment files.

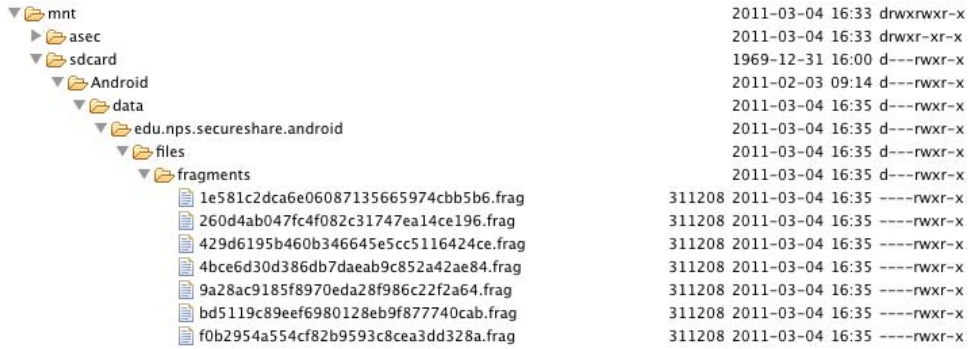

Figure 4.3: SecureShare Fragments Directory

To verify that MDFS does indeed encrypt the data as expected, we ran MDFS on a text file and inspected the resulting fragments generated by plain erasure coding, using JErasure 1.2, and MDFS. Figure 4.4 shows the first few lines of a text file called definitions.txt. Figure 4.5 shows the same file as viewed in a HEX editor. We used a HEX editor because the encrypted fragment is not viewable in a standard text editor and we are able to compare both the text view and the corresponding HEX values. We erasure encoded definitions.txt using a value of $n = 5$ and $k = 3$. Figure 4.6 is a view of the first fragment generated by JErasure 1.2. Fragment 1 contains the first one third of definitions.txt, so we expect the beginning of definitions.txt and fragment 1 to be identical. Next we encoded definitions.txt with MDFS using the same n and k values. Figure 4.7 shows fragment 1 of MDFS. Fragment 1 generated by MDFS contains the same information, but is AES encrypted and therefore no information about the actual contents of definitions.txt can be determined by simply inspecting the fragment.

The purpose of this analysis is not to demonstrate robust defense against a cryptanalysis attack on MDFS, but rather to show that MDFS does provide security against simple inspection of fragments as intended. The ability to withstand sophisticated cryptanalysis is left for future work.

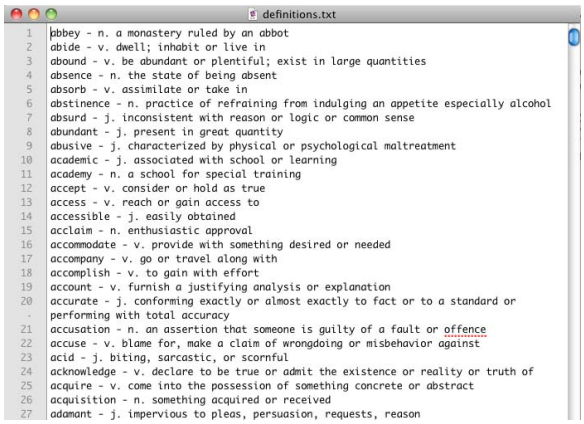

Figure 4.4: definitons.txt File

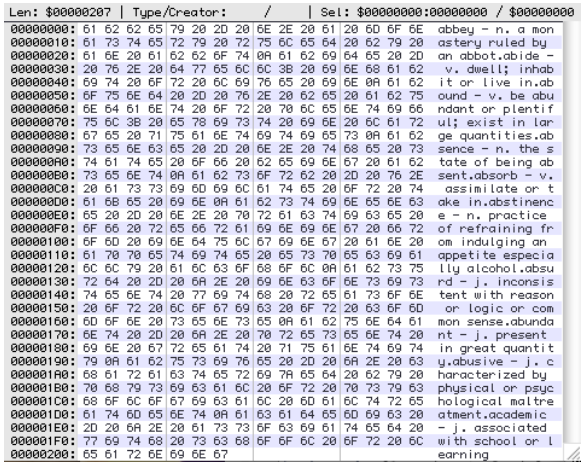

Figure 4.5: HEX View of definitions.txt

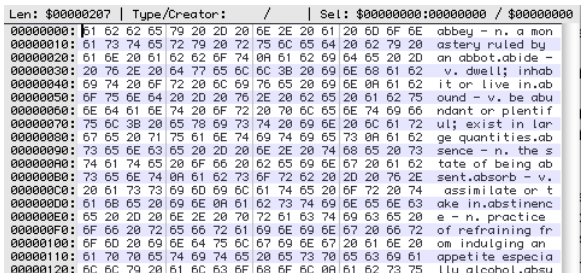

Figure 4.6: HEX View of Fragment 1 (Erasure Code)

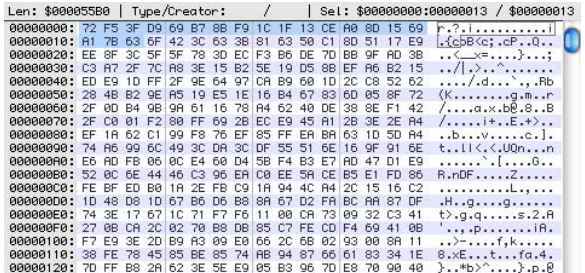

Figure 4.7: HEX View of Fragment 1 (MDFS)

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 5:
Future Work and Conclusion

5.1 Fragment Regeneration

Once issue that arises when utilizing erasure coding for distributed storage is the need to replace fragments due to loss. Erasure coding provides some inherent redundancy, however as nodes leave or become otherwise unreachable, resiliency is diminished. One naive, yet inefficient, method of providing redundancy is to store multiple copies of each fragment at different nodes thus increasing the storage complexity and space requirement and network overhead while reducing efficiency. For example, the system could store n fragments to $2n$ devices so there are 2 complete copies of all the fragments in the network. If one is lost, a copy can be made from the remaining fragment to ensure two copies are always available. Another method is for the directory service to periodically poll each node to ensure it is reachable and if not, regenerate fragments by recovering the original file and redistributing the new fragments. This is also an incredibly inefficient design. Dimakis presents Regenerating Codes which address this issue by providing a more efficient way to regenerate lost fragments under certain conditions [10].

5.2 Device Authentication

The goal of MDFS is to provide security and resiliency, but makes the assumption that authentication is handled external to MDFS. It is assumed that physical access to the fragments is sufficient authentication. With the current implementation it is important that sufficient wireless security be implemented to prevent fragments from being gathered by passive snooping. The fragments themselves are symmetrically encrypted, but MDFS provides no mechanism to prevent a third party from gathering enough packets through passive sniffing to reconstruct the symmetric key and decode the original file.

One way to mitigate this problem would be to implement public key infrastructure (PKI) encryption on top of MDFS. During the initial storage operation, the content generator could request public keys from each of the destination nodes and encrypt the fragments using the public key to secure the fragment in transit. During the read operation, the content consumer could include its own public key in the request fragment message. As previously discussed, asymmetric encryption is a relatively expensive operation, so the encrypting node could use the destination's public key to encrypt just the symmetric key fragment as the fragment itself

is already encrypted. Therefore, the symmetric key can only be decoded by the requestor, and consequently, the entire fragment maintains its security.

5.3 File Size Limitation

The current implementation has a limitation on the size of files that can be processed. Heap memory is limited on mobile architectures such as Android and iOS. In most modern Android devices, each application is limited to 16 - 24MB of memory. MDFS implements the erasure code in a native library and when the native code is called, an array of byte arrays is passed back that is larger than the original file. System instability results if the array of byte arrays exceeds the size allowed by the operating system. This condition can be easily triggered by setting the k value small and the n value large. The size of the array of byte arrays is $(\text{filesize}/k) \times n$. When the native library passes an array that is too large for the Android heap, SecureShare quits with no error message or exception thrown.

The alternative solution would be to pass back each fragment as an individual byte array one at a time, but that would require a rewrite of the native code. It would require the native code to write the byte array directly to the filesystem on the android device as it is generated. Revision 5 of the NDK supports native calls to Android methods, so this approach may be more feasible than it was prior to this release.

5.4 Cryptographic Security

Section 4.6.4 was intended to show that MDFS provides encryption that will withstand simple inspection of the generated fragments. MDFS has not been subjected to extensive forensic analysis or sophisticated cryptanalysis. Study and feedback from experienced security experts would be useful for ensuring the cryptographic security of MDFS.

5.5 Secure Data Deletion

Filesystem permissions are very tightly controlled by the Android OS. Access to images and videos are regulated by Content Providers and Media Controllers. File access permissions are controlled by the physical storage location. The data fragments are stored on the external media card storage of the Android device. Android's security permissions prevent standard Android methods from accessing files once they are deleted, however the susceptibility of data to forensic techniques is outside the scope of this work. In order to secure the data against a logical device

dump, it will probably be necessary to implement secure delete methods in native libraries capable of accessing the file system below the Android layer.

5.6 Distributed Directory Service

This thesis proposes a distributed directory service, but implemented the directory service as a well known host on one of the phones. This approach achieved the goal of running MDFS without the need of extensive infrastructure, but fell short of the goal to not have a single point of failure. While the SecureShare directory service runs on a single node, even the hosting node accesses the directory service through the network API. The application installed on the directory service node is identical to the installation on all the other nodes.

The move from the single node directory service to a distributed directory service will involve the implementation of an additional service on the phone. The current service receives and processes unicast messages. In order to implement a completely distributed directory service, it will be necessary to add an additional service that sends and receives messages on a multicast channel. The most challenging issue for maintaining a distributed directory service will be managing database synchronization between the various nodes. Each node must have the most up-to-date information about which files and fragments are available in the system.

The Directory Service is implemented as a SQLite database that keeps track of the filenames and where they are stored. Fragment filenames are created as a hash of the original filename, the timestamp of the original file, and the fragment number. One important function that needs to be added is a delete function. The process is the opposite of the write function. First the filename needs to be removed from the files database to prevent other nodes from requesting the file. Then a message needs to be sent to each node that stores a fragment of that file instructing it to delete the fragment and the fragment hashes need to be purged from the fragments database.

5.7 Metrics

The implementation of MDFS resulting from this thesis functioned adequately for images. It was able to transmit fragments and recover images without noticeable lag or significant delay, but throughput metrics were not gathered. A detailed study of throughput and performance metrics needs to be conducted. Simple performance metrics on MDFS fragment generation and file recovery would be a good start to determine if the mobile devices are capable of performing the necessary erasure coding and Shamir coding within a reasonable threshold. This metric excludes file I/O and network latency. A more detailed analysis of performance should be

conducted on a full scale deployment of MDFS with multiple devices where total throughput, including file I/O and network delays, can be gathered.

5.8 Conclusion

This implementation of MDFS, while primitive and in its very early stage, shows that mobile phones have the processing power and functionality to support the requirements of MDFS. There is still much work to be done before this is a viable system for deployment. We feel that the most severe limitation is that the underlying directory service is unable to function when the phone is sleeping. While it may be possible to mitigate the effects of this problem by implementing the directory service as a delay tolerant network (DTN), that will, by definition, introduce significant delays to the system.

MDFS successfully combines erasure coding, Shamir's secret sharing algorithm, AES symmetric encryption, and a directory service to store files securely in a distributed file system running on a mobile wireless network. It meets the requirements as set forth in Section 1.2. MDFS can tolerate the loss of up to $n - k$ nodes and still recover data.

REFERENCES

[1] J. S. Dixon, "Integrating cellular handset capabilities with marine corps tactical communications," Master's thesis, Naval Postgraduate School, Monterey, CA, 2010.

[2] F. Chang, J. Dean *et al.*, "Bigtable: a distributed storage system for structured data," in *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, Berkeley, CA, 2006, pp. 205–218.

[3] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communication of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[4] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, ser. SOSP '03, New York, NY, USA, 2003, pp. 29–43.

[5] Z. O'Whielacronx. (2011) zfec 1.4.22 erasure codec. [Online]. Available: http://pypi.python.org/pypi/zfec

[6] Tahoe-LAFS. (2010) Tahoe least authority file system. [Online]. Available: http://tahoe-lafs.org/trac/tahoe-lafs

[7] P. Zimmerman and W. Stallings, "Pgp message exchange formats," RFC 1991, Aug. 1996.

[8] *Advanced Encryption Standard (AES)*, NIST Std. FIPS 197, 2001.

[9] I. Reed and G. Solomon, "Polynomial codes over certain finite fields," *Journal of the Society for Industrial and Applied Mathematics*, vol. 8, pp. 300–304, Jun 1960.

[10] A. Dimakis, P. Godfrey, Y. Wu, M. Wainwright, and K. Ramchandran, "Network coding for distributed storage systems," *IEEE Transactions on Information Theory*, vol. 56, pp. 4539 – 4551, Sep. 2010.

[11] D. Borthakur. (2009) Implement erasure coding as a layer on HDFS. [Online]. Available: https://issues.apache.org/jira/browse/HDFS-503

[12] R. L. Rivest, A. Shamir, and Y. Tauman, "How to share a secret," *Communication of the ACM*, 1979.

[13] G. Singh, “Information sharing for emergency response,” in *2008 IEEE Conference on Technologies for Homeland Security*, 2008, pp. 421 –425.

[14] S. L. Flosi. (2011) comScore reports january 2011 U.S. mobile subscriber market share. [Online]. Available: http://www.comscore.com/Press_Events/Press_Releases/2011/3/comScore_Reports_January_2011_U.S._Mobile_Subscriber_Market_Share

[15] Google. (2011) Android ndk. [Online]. Available: http://developer.android.com/sdk/ndk/index.html

[16] J. S. Plank, J. Luo, C. D. Schuman, L. Xu, and Z. Wilcox-O’Hearn, “A performance evaluation and examination of open-source erasure coding libraries for storage,” in *FAST-2009: 7th Usenix Conference on File and Storage Technologies*, February 2009.

[17] T. Tiemens. (2011) Shamir secret sharing in java. [Online]. Available: http://sourceforge.net/projects/secretsharejava/

[18] B. Schneier, *Applied Cryptography*. New York, NY: John Wiley & Sons, Inc., 1996.

Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California